K and Matching Logic

Grigore Rosu
University of Illinois at Urbana-Champaign

Joint work with the FSL group at UIUC (USA)
and the FMSE group at UAIC (Romania)

Question

… could it be that, after 40 years of program verification, we still lack the right semantically grounded program verification foundation?

?

Hoare logic

$$\{\pi_{\mathsf{pre}}\}\ \mathsf{code}\ \{\pi_{\mathsf{post}}\}$$

Current State-of-the-Art in Program Analysis and Verification

Consider some programming language, L

- Formal semantics of L?
  - Typically skipped: considered expensive and useless
- Model checkers for L
  - Based on some adhoc encodings/models of L
- Program verifiers for L
  - Based on some other adhoc encodings/models of L
- Runtime verifiers for L
  - Based on yet another adhoc encodings/models of L
- …

Example of C Program

- What should the following program evaluate to?

```
int main(void) {
  int x = 0;
  return (x = 1) + (x = 2);
}
```

- According to the C “standard”, it is **undefined**
- GCC4, MSVC: it returns **4**
  GCC3, ICC, Clang: it returns **3**
  By April 2011, both Frama-C (with its Jessie verification plugin) and Havoc "prove" it returns **4**

A Formal Semantics Manifesto

- Programming languages must have formal semantics**!** (period)
  - And analysis/verification tools should build on them at best, or should formally relate to them at worst
    - Otherwise they are adhoc and likely wrong

- Informal manuals are not sufficient
  - Manuals typically have a formal syntax of the language (in an appendix)
  - Why not a formal semantics appendix as well?

Motivation and Goal

- We are facing a semantic chaos
  - Axiomatic, denotational, operational, etc.
- Why so many semantic styles?
  - Since none of them is ideal, they have limitations

- We want a powerful, unified foundation for language design, semantics and verification
  - One semantic approach to serve all the purposes!
  - To work with realistic languages (C, Java, etc.)

Minimal Requirements for an Ideal Language Semantic Framework

- Should be expressive
  - Substitution or environment-based definitions, abrupt control changes (callcc), concurrency, etc.
- Should be (efficiently) executable
  - So we can test it and use it in tools (symb. exec.)
- Should be modular (thus scale)
  - So each feature is defined once and for all
- Should serve as a basis for program reasoning
  - So we can also prove programs correct with it

- Conventional Semantic Approaches -
- Advantages and Limitations –

Chronologically

- 1969: Floyd-Hoare Logic -

- Basis for program verification
- Not easily executable, and thus, hard to test
  - Semantic errors found by proving wrong properties
  - Soundness rarely or never proved in practice
- Not very expressive
  - Often requires heavy program transformations (e.g., to eliminate side effects, pointers, exceptions, etc.), to reduce languages to cores which can be given an axiomatic semantics
  - Structural program properties (e.g., about heap, stacks, input/output, etc.) hard to state; need special logic support
  - Structural framing (e.g., heap framing) hard to deal with
- Implementations of Floyd-Hoare verifiers for real languages still an art, who few master

- 1971: Denotational Semantics -

Reasonable trade-offs. “Compiles” programs into mathematical objects, so it can be *in principle*:

- Expressive, provided enough/appropriate mathematical domains available
- Executable: we can execute/approximate fixed-points
  - Although factorial(5) crashes Papaspyrou’s C semantics
- Modular, provided one uses advanced features
  - Monads, continuations, resumptions
- Basis for program verification
  - Program = least fixed point, so we can use induction
- Hard to use and understand; requires expert knowledge; no overwhelming evidence it is practical for verification

- 1981: Operational Semantics -

Quite intuitive, easy to understand and define.
Requires minimal training and it scales.

- Executable, by its very nature
  - Although Norish's C semantics not executable, evaluation contexts are inefficient, CHAM has no machine support, etc.
- Expressive … in principle
  - Although hard to use both evaluation contexts (for call/CC, longjumps,…) and environment-store (for pointers, threads,…)
- Modular, when one uses
  - MSOS ideas for dealing with configuration changes, evaluation contexts ideas for control, CHAM ideas for concurrency, etc.
- Considered "too low level", inappropriate for verification…

What We Want

- First, we want a semantic framework which, at the same time and uniformly well, is
  - Expressive, Executable, Modular, and
  - Suitable for program reasoning
- Second, we want to develop supporting tools for
  - Defining formal language semantics, and
  - Using the semantics for program verification
    - Put an end to having both operational and axiomatic or other semantics to languages. No more semantics equivalence proofs to be done and maintained!

Our quest

Next I will tell a story about our quest for such a unified and practical semantic framework

Take our approach as a possibility …

… not as ultimate answer

Message to take home:

This is not a dream!

Starting Point: Rewriting Logic

Meseguer (late 80s, early 90s)

- Expressive
  - Any logic can be represented in RL (it is reflective)
- Executable
  - Quite efficiently; Maude often outperforms SML
- Modular
  - Allows rules to only “match” what they need
- Can serve as a basis for program reasoning
  - Admits initial model semantics, so it is amenable for inductive or fixed-point proofs

The K Framework

`k-framework.org`

- A tool-supported rewrite-based framework for defining programming language semantics
- Main ideas
  - Represent program configurations as potentially nested structures of cells (like in the CHAM)
  - Flatten syntax into special computational structures (like in refocusing for evaluation contexts)
  - Define the semantics of each language construct by semantic rules (a small number, typically 1 or 2)

Complete K Definition of KernelC

MODULE KERNELC-SYNTAX
IMPORTS K-LATEX+PL-ID+PL-INT
SYNTAX *Exp* ::= *Exp* + *Exp* [strict]
| *DeclId*
| *Id*
| *Int*
| *Exp* - *Exp* [strict]
| *Exp* ++
| *Exp* == *Exp* [strict]
| *Exp* != *Exp* [strict]
| *Exp* <= *Exp* [strict]
| *Exp* < *Exp* [strict]
| *Exp* % *Exp* [strict]
| ! *Exp*
| *Exp* && *Exp*
| *Exp* ? *Exp* : *Exp*
| *Exp* || *Exp*
| printf("%d;", *Exp*) [strict]
| scanf("%d",& *Exp*)
| scanf("%d", *Exp*) [strict]
| NULL
| *PointerId*
| (int∗)malloc(*Exp* ∗sizeof(int)) [strict]
| free(*Exp*) [strict]
| ∗ *Exp* [strict]
| *Exp* [*Exp*]
| *Exp* = *Exp* [strict(2)]
| *Id* (*List{Exp}*) [strict(2)]
| *Id* ()
| random()
| srandom(*Exp*) [strict]
SYNTAX *Stmt* ::= *Exp* ; [strict]
| {}
| { *StmtList* }
| if(*Exp*) *Stmt*
| if(*Exp*) *Stmt* else *Stmt* [strict(1)]
| while(*Exp*) *Stmt*
| return *Exp* ; [strict]
| *DeclId* *List{DeclId}* { *StmtList* }
| #include< *StmtList* >
SYNTAX *StmtList* ::= *StmtList* *StmtList*
| *Stmt*
SYNTAX *Pgm* ::= *StmtList*
SYNTAX *Id* ::= main
SYNTAX *PointerId* ::= ∗ *PointerId* [ditto]
| *Id*
SYNTAX *DeclId* ::= int *Exp*
| void *PointerId*
SYNTAX *StmtList* ::= stdio.h
| stdlib.h
SYNTAX *List{Bottom}* ::= *List{Bottom}* , *List{Bottom}* [assoc hybrid id: () strict]
| ()
| *Bottom*
SYNTAX *List{PointerId}* ::= *List{PointerId}* , *List{PointerId}* [ditto]
| *List{Bottom}*
| *PointerId*
SYNTAX *List{DeclId}* ::= *List{DeclId}* , *List{DeclId}* [ditto]
| *DeclId*
| *List{Bottom}*
SYNTAX *List{Exp}* ::= *List{Exp}* , *List{Exp}* [ditto]
| *Exp*
| *List{DeclId}*
| *List{PointerId}*
END MODULE

MODULE KERNELC-DESUGARED-SYNTAX
IMPORTS K-LATEX
IMPORTS KERNELC-SYNTAX
MACRO ! E = E ? 0 : 1

MACRO E_1 && E_2 = E_1 ? E_2 : 0

MACRO E_1 || E_2 = E_1 ? 1 : E_2

MACRO if(E) *St* = if(E) *St* else {}

MACRO NULL = 0

MACRO I () = I (())

MACRO int ∗ *PointerId* = int *PointerId*

MACRO #include< *Stmts* > = *Stmts*

MACRO E_1 [E_2] = ∗ E_1 + E_2

MACRO scanf("%d",& ∗ E) = scanf("%d", E)

MACRO int ∗ *PointerId* = E = int *PointerId* = E

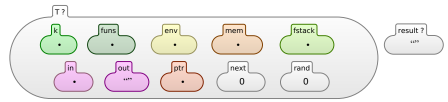
MACRO int X = E ; = int X ; X = E ;

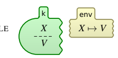
MACRO stdio.h = {}

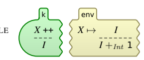
MACRO stdlib.h = {}

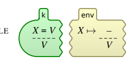
END MODULE

MODULE KERNELC-SEMANTICS
IMPORTS K-SHARED
IMPORTS K+KERNELC-DESUGARED-SYNTAX+PL-CONVERSION+PL-RANDOM
CONFIGURATION:



RULE 

RULE 

RULE 

RULE I_1 + I_2 ⇀ I_1 $+_{Int}$ I_2

RULE I_1 - I_2 ⇀ I_1 $-_{Int}$ I_2

RULE I_1 % I_2 ⇀ I_1 $\%_{Int}$ I_2 when I_2 $!=_{Int}$ 0

RULE I_1 <= I_2 ⇀ Bool2Int ($I_1 \leq_{Int} I_2$)

RULE I_1 < I_2 ⇀ Bool2Int ($I_1 <_{Int} I_2$)

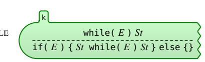
RULE I_1 == I_2 ⇀ Bool2Int (I_1 $==_{Int}$ I_2)

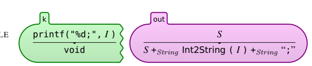
RULE I_1 != I_2 ⇀ Bool2Int (I_1 $!=_{Int}$ I_2)

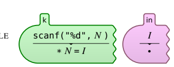
RULE _?_:_ ⇀ if(_)_else_

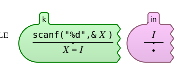
RULE if(I) – else *St* ⇀ *St* when I $==_{Int}$ 0

RULE if(I) *St* else – ⇀ *St* when $\neg_{Bool}$ I $==_{Int}$ 0

RULE 

RULE 

RULE 

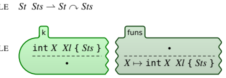
RULE 

RULE V ; ⇀ •

RULE { *Sts* } ⇀ *Sts*

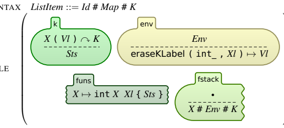
RULE {} ⇀ •

RULE *St* *Sts* ⇀ *St* ↷ *Sts*

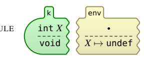
RULE 

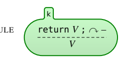
RULE $\underset{\texttt{int}\ X}{\underline{\texttt{void}\ X}}$ *Xl* { $\underset{Sts\ \texttt{return void ;}}{\underline{Sts}}$ }

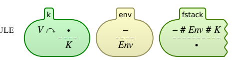
SYNTAX *ListItem* ::= *Id* # *Map* # *K*

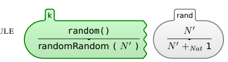
RULE 

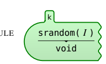
CONTEXT: int – = □

RULE 

RULE 

RULE 

RULE 

RULE 

CONTEXT: ∗ □ = –

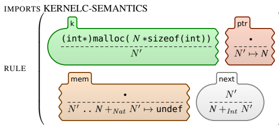
CONTEXT: ∗ □ ++

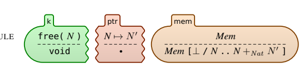
SYNTAX *Val* ::= *Int*
| void
SYNTAX *Exp* ::= *Val*
SYNTAX *K* ::= *List{DeclId}*
| *List{Exp}*
| *List{PointerId}*
| *Pgm*
| *StmtList*
| *String*
| restore(*Map*)
| undef
SYNTAX *KResult* ::= *List{Val}*
SYNTAX *List{K}* ::= *Nat* .. *Nat*
RULE N_1 .. N_1 ⇀ •

RULE N_1 .. s_{Nat} N ⇀ N , N_1 .. N

SYNTAX *List{Val}* ::= *List{Val}* , *List{Val}* [ditto]
| *Val*
SYNTAX *List{Exp}* ::= *List{Val}*
END MODULE

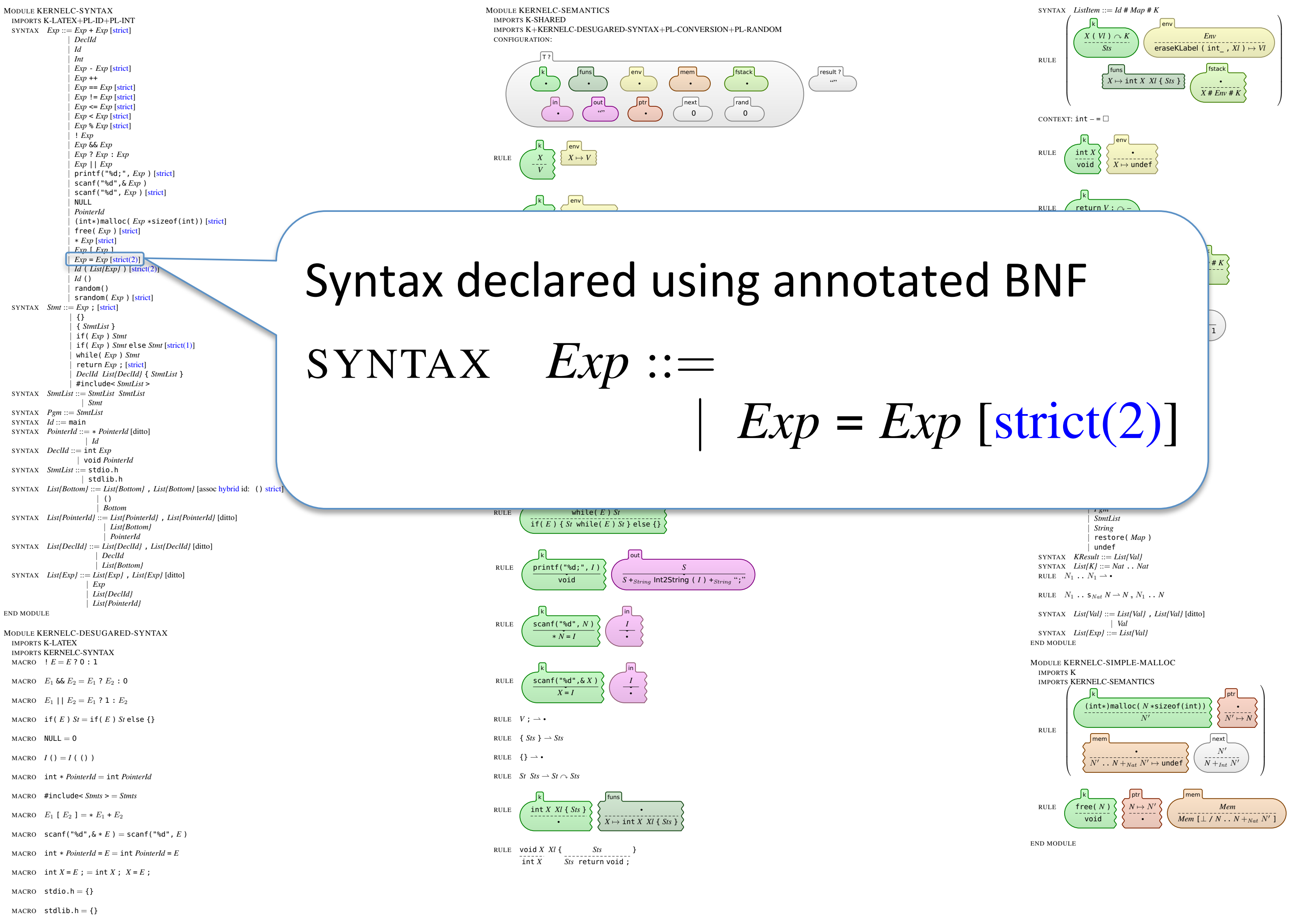
MODULE KERNELC-SIMPLE-MALLOC
IMPORTS K
IMPORTS KERNELC-SEMANTICS

RULE 

RULE 

END MODULE

Complete K Definition of KernelC



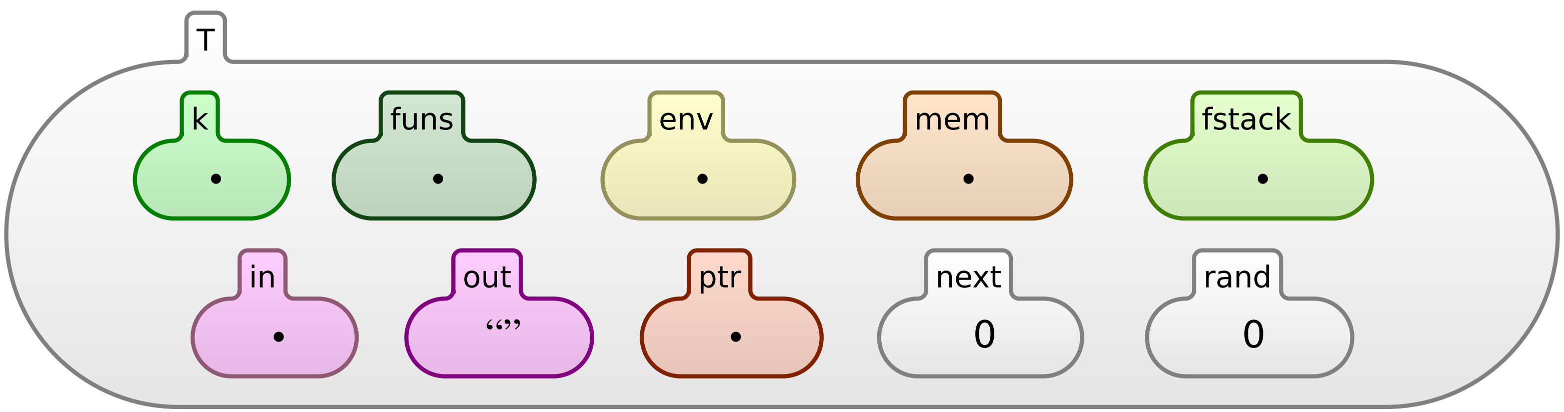
Syntax declared using annotated BNF

SYNTAX $\quad Exp ::=$

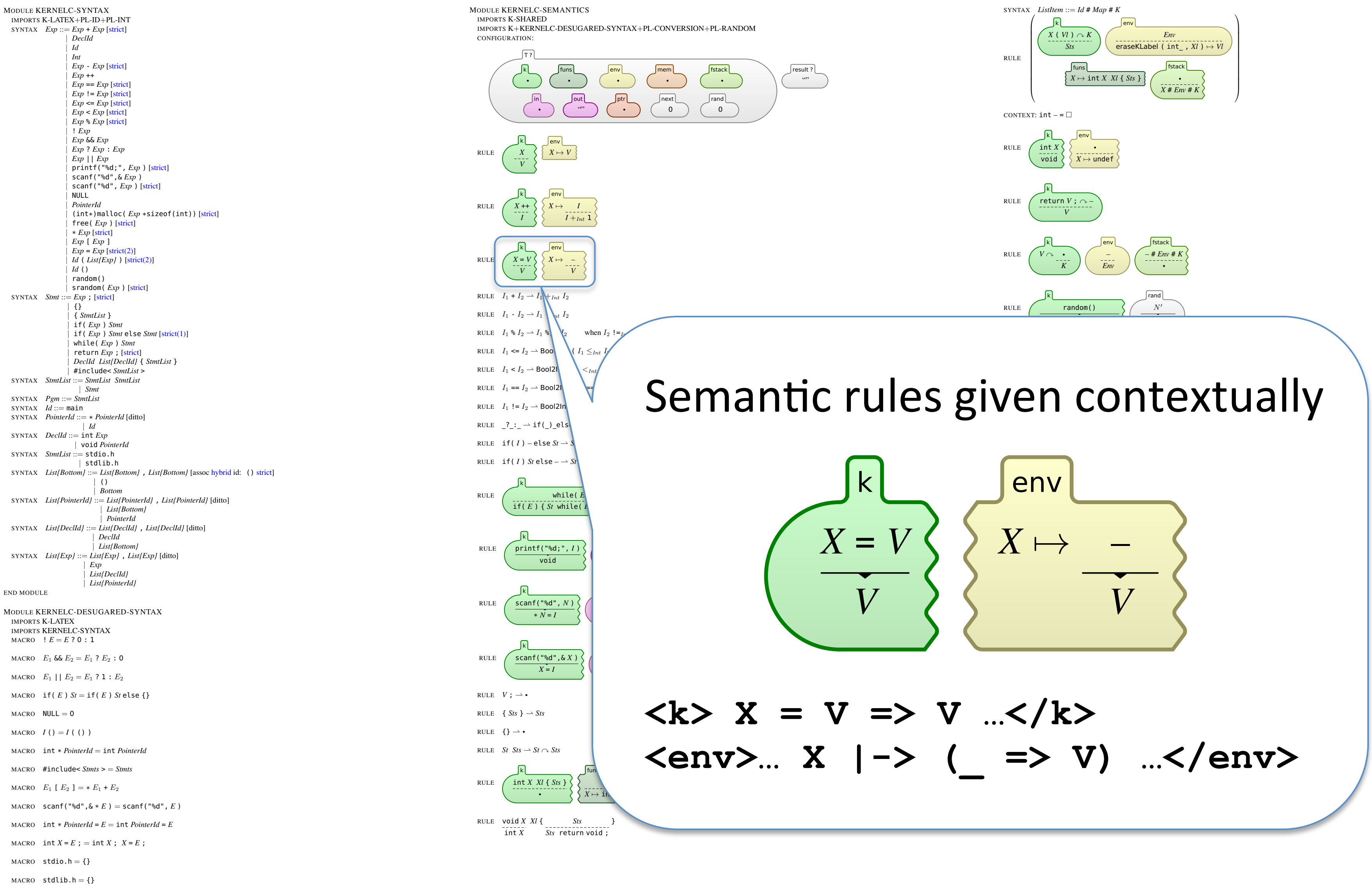
$\qquad\qquad\mid\ Exp = Exp$ [strict(2)]

Complete K Definition of KernelC

Configuration given as a nested cell structure.
Leaves can be sets, multisets, lists, maps, or syntax



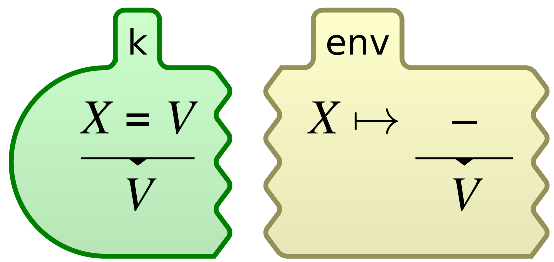
Complete K Definition of KernelC



Don’t Like Bubbles?

- The KernelC definition above was generated by the K tool using the bubble mode (**`bb`**)

`bb`
mode



- K tool also provides a mathematical mode (**`mm`**), which may be preferred in formal writing

`mm`
mode

$$\left\langle \frac{X = V}{V} \cdots \right\rangle_{\mathsf{k}} \left\langle \cdots X \mapsto \frac{\_}{V} \cdots \right\rangle_{\mathsf{env}}$$

What is K, after all … ?

Except for its true concurrency semantics, based on graph rewriting, K is a technique and notation to define languages as rewrite systems with rules

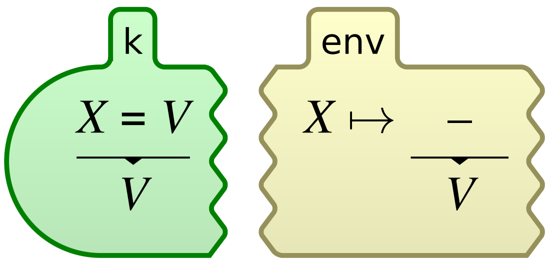
$$l \Rightarrow r \text{ if } b$$

(*b* is a side condition)

... and so are reduction semantics with evaluation contexts, (chemical) abstract machines, etc.

Translating K into rules $l \Rightarrow r \text{ if } b$

SYNTAX $\quad Exp ::= Exp = Exp$ [strict(2)]



kompile into:

$$\text{RULE} \quad E_1{=}E_2 \quad \Rightarrow \quad E_2 \curvearrowright E_1{=}\square \quad \text{if} \quad E_2 \notin Val$$
$$\text{RULE} \quad E_2 \curvearrowright E_1{=}\square \quad \Rightarrow \quad E_1{=}E_2 \quad \text{if} \quad E_2 \in Val$$

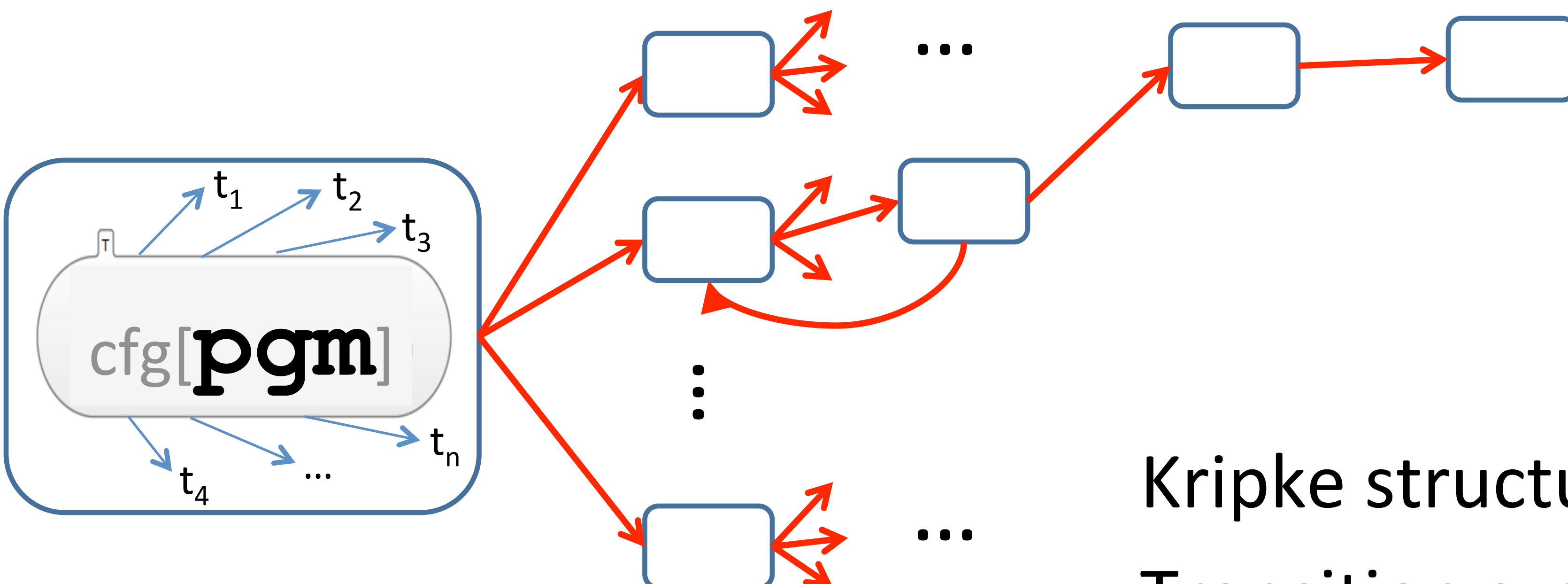
$$\text{RULE} \quad \mathsf{k}(X{=}V \curvearrowright K)\ \mathsf{env}(\rho_1,\ X \mapsto V',\ \rho_2)$$
$$\Rightarrow \mathsf{k}(\ V \ \curvearrowright K)\ \mathsf{env}(\rho_1,\ X \mapsto V,\ \rho_2)$$

What is the K Semantics of a Program?

Two types of K rules:

Structural: rearrange configuration, unobservable

Computational: count as computational steps

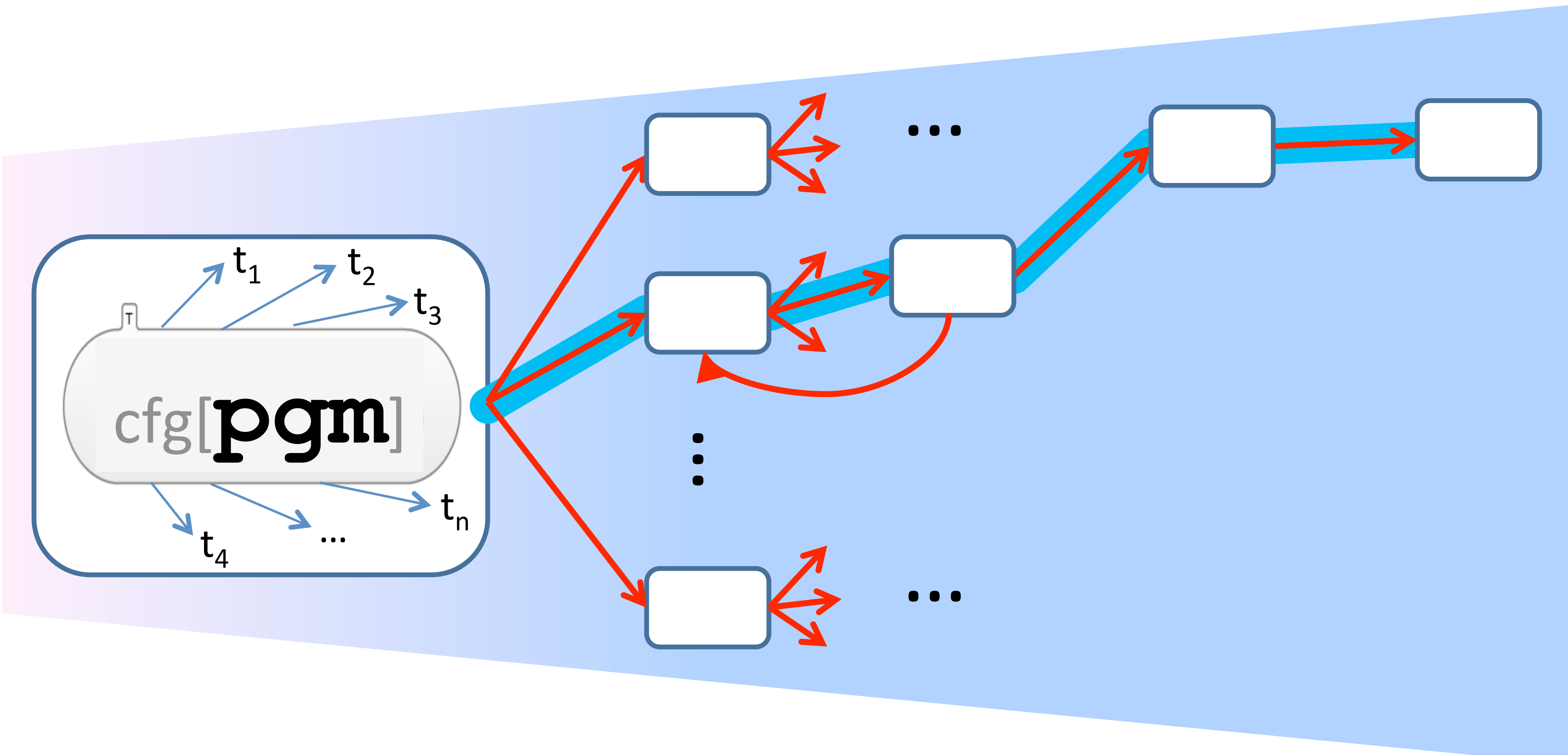


t_1, …, t_n = structural variants

Kripke structure

Transition system

What does the K Tool Offer?



Efficient and interactive execution (interpreters)

State-space exploration (search and model-checking)

K Semantics are Useful

- Executable, help language designers
- Make teaching PL concepts hands-on and fun
- Currently compiled into
  - Maude, for execution, debugging, model checking
  - Latex, for human inspection and understanding
- Plans to be compiled to
  - OCAML, for fast execution
  - COQ, for meta-property verification

Medium-Size K Definition

See the

- dynamic semantics or the
- type checker

of SIMPLE

K Scales

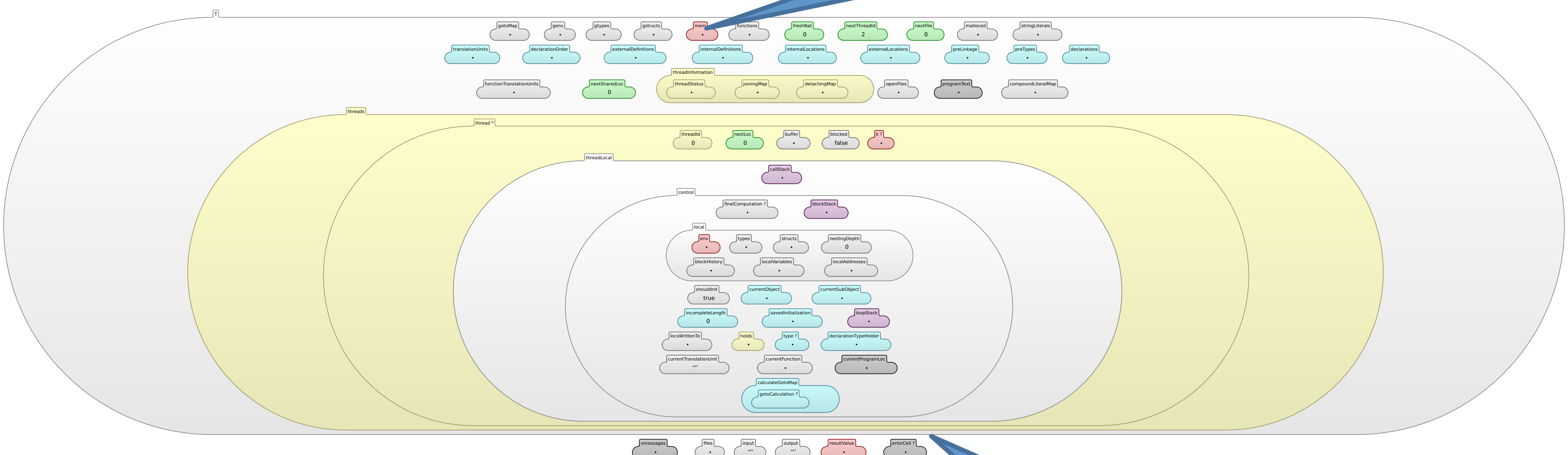
Besides smaller and paradigmatic teaching languages, several larger languages were defined

- Scheme : by Pat Meredith
- Java 1.4 : by Feng Chen
- Verilog : by Pat Meredith and Mike Katelman
- C : by Chucky Ellison

etc.

The K Configuration of C

Heap



75 Cells!

Statistics for the C definition

- Total number of rules: **~1200**
- Has been tested on thousands of C programs (several benchmarks, including the gcc torture test, code from the obfuscated C competition, etc.)
  - Passed **99.2%** so far!
  - GCC 4.1.2 passes 99%, ICC 99.4%, Clang 98.3 (no opt.)
- *The most complete formal C semantics*
- Took more than 18 months to define …
  - Wouldn't it be uneconomical to redefine it in each tool?

Matching Logic Verification

=

Rewriting (language semantics)

+

[FOL] (configuration reasoning)

+

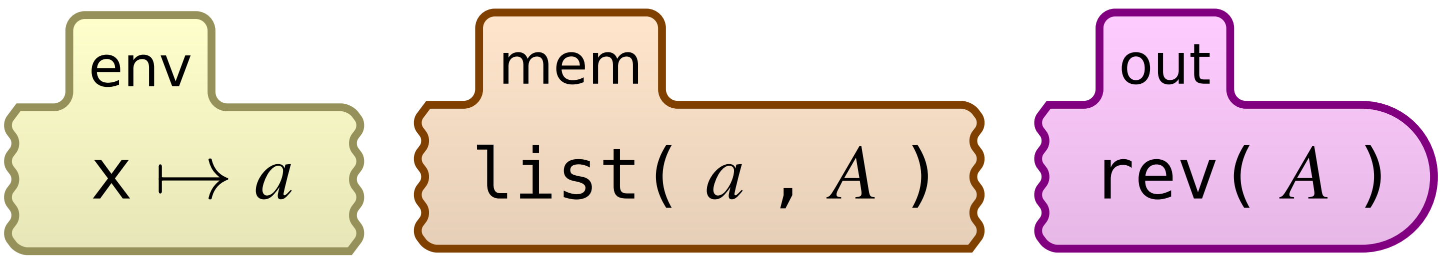
Proof Rules (behavior reasoning)

Matching Logic

- A logic for reasoning about configurations
- Formulae
  - [FOL] over configurations, called patterns
  - Configurations are allowed to contain variables
- Models
  - Ground configurations
- Satisfaction
  - Matching for configurations, plus [FOL] for the rest

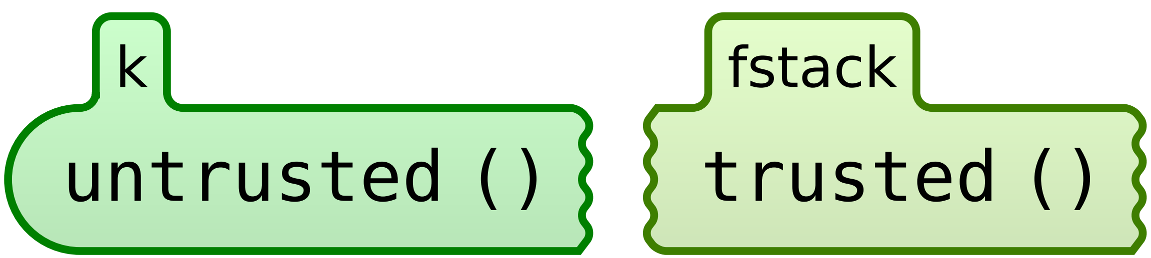
Examples of Patterns

- x points to sequence *A* with |*A*|>1, and the reversed sequence rev(*A*) has been output



$\land\ |A|>1$

- **`untrusted()`** can only be called from **`trusted()`**



More Formally: Configurations

- For concreteness, assume configurations having the following syntax:

$$\langle\langle\ldots\rangle_{\mathsf{k}}\ \langle\ldots\rangle_{\mathsf{env}}\ \langle\ldots\rangle_{\mathsf{heap}}\ \langle\ldots\rangle_{\mathsf{in}}\ \langle\ldots\rangle_{\mathsf{out}}\ \ \ldots\rangle_{\mathsf{cfg}}$$

(matching logic works with any configurations)

- Examples of concrete (ground) configurations:

$$\langle\langle \mathtt{x{=}{*}y\,;\ y{=}x\,;}\ \ldots\rangle_{\mathsf{k}}\ \langle \mathtt{x} \mapsto 7,\ \mathtt{y} \mapsto 3,\ \ldots\rangle_{\mathsf{env}}\ \langle 3 \mapsto 5\rangle_{\mathsf{heap}}\ \ \ldots\rangle_{\mathsf{cfg}}$$

$$\langle\langle \mathtt{x} \mapsto 3\rangle_{\mathsf{env}}\ \langle 3 \mapsto 5,\ 2 \mapsto 7\rangle_{\mathsf{heap}}\ \langle 1, 2, 3, \ldots\rangle_{\mathsf{in}}\ \langle \ldots, 7, 8, 9\rangle_{\mathsf{out}}\ \ \ldots\rangle_{\mathsf{cfg}}$$

More Formally: Patterns

- Concrete configurations are already patterns, but very simple ones, ground
- Example of more complex pattern

$$\exists c : Cells,\ e : Env,\ p : Nat,\ \ i : Int,\ \sigma : Heap$$
$$\underline{\langle\langle \mathbf{x} \mapsto p,\ e\rangle_{\mathsf{env}}\ \langle p \mapsto i,\ \sigma\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}}}\ \wedge\ i > 0\ \wedge\ p \neq i$$

- Thus, patterns generalize both terms and [FOL]
- Models: concrete configurations + valuations
- Satisfaction: matching for patterns, [FOL] for rest

More Formally: Reasoning

- We can now prove (using FOL reasoning) properties about configurations, such as

$$
\begin{aligned}
\models \quad & \forall c\!:\!Cell,\ e\!:\!Env,\ p\!:\!Nat \\
& \quad \langle\langle \mathbf{x} \mapsto p,\ e\rangle_{\mathsf{env}}\ \langle p \mapsto 9\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}}\ \wedge\ p > 10 \\
& \quad \to \exists i\!:\!Int,\ \sigma\!:\!Heap \\
& \qquad\qquad \langle\langle \mathbf{x} \mapsto p,\ e\rangle_{\mathsf{env}}\ \langle p \mapsto i,\ \sigma\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}}\ \wedge\ i > 0\ \wedge\ p \neq i
\end{aligned}
$$

Matching Logic vs. Separation Logic

- Matching logic achieves separation through matching at the structural (term) level, not through special logical connectives (*)
- Matching logic realizes separation at all levels of the configuration, not only in the heap
  - the heap was only 1 out of the 75 cells in C's def.
- Matching logic can stay within FOL, while separation logic needs to extend FOL
  - Thus, we can use the existing SMT provers, etc.

Matching Logic Verification

- Hoare style - not recommended

$$\{\pi_{\mathsf{pre}}\}\ \mathsf{code}\ \{\pi_{\mathsf{post}}\}$$

  – Need to redefine the PL semantics – impractical

- Rewriting (or K) style – recommended

$$left[\mathsf{code}] \rightarrow right$$

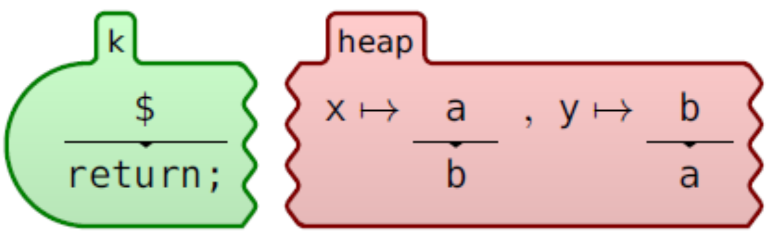
  – Can reuse existing operational semantics – good

Example – Swapping Values

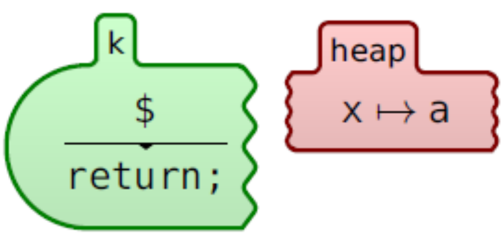
$

```
void swap(int *x, int *y)
{
  int t;
  t=*x;
  *x=*y;
  *y=t;
}
```

- What is the K semantics of the swap function?
- Let $ be its body




```
rule <k> $ => return; …</k>
     <heap>…
       x|->(a=>b),
       y|->(b=>a)
     …</heap>
```



if x = y

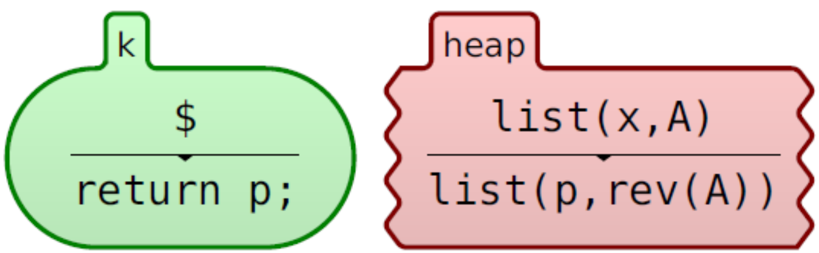
```
rule <k> $ => return; …</k>
     <heap>… x|-> a …</heap>
  if x = y
```

Example – Reversing a list

$

```
struct listNode* reverse(struct listNode *x)
{
  struct listNode *p;
  struct listNode *y;
  p = 0 ;
  while(x) {
    y = x->next;
    x->next = p;
    p = x;
    x = y;
  }
  return p;
}
```

- What is the K semantics of the reverse function?
- Let $ be its body




```
rule <k> $ => return p; </k>
     <heap>… list(x,A) => list(p,rev(A)) …</heap>
```

Partial Correctness

- We have two rewrite relations on configurations
  - $\rightarrow$ given by the language K semantics; safe
  - $\rightarrow$ given by specifications; unsafe, has to be proved
- Idea (simplified for deterministic languages):
  - Pick left $\rightarrow$ right. Show that always left $\rightarrow$ $(\rightarrow \cup \rightarrow)^*$ right modulo matching logic reasoning (between rewrite steps)

- Theorem (soundness):
  - If left $\rightarrow$ right and "config *matches* left" such that config has a normal form for $\rightarrow$, then "nf(config) *matches* right"

More Formally: Matching Logic Rewriting

- Matching logic rewrite rules are rewrite rules over matching logic formulae: $\varphi \Rightarrow \varphi'$
- Since patterns generalize terms, matching logic rewriting captures term rewriting
- Moreover, deals naturally with side conditions: rewrite rules of the form

$$l \Rightarrow r \text{ if } b$$

are captured as matching logic rules of the form

$$l \wedge b \Rightarrow r$$

More Formally: Proof System I

- Rules of operational nature

Reflexivity

$$\frac{\cdot}{\mathcal{A} \vdash \varphi \Rightarrow \varphi}$$

Axiom

$$\frac{\varphi \Rightarrow \varphi' \in \mathcal{A}}{\mathcal{A} \vdash \varphi \Rightarrow \varphi'}$$

More Formally: Proof System II

Substitution

$$\frac{\mathcal{A} \vdash \varphi \Rightarrow \varphi' \qquad \theta : Var \to \mathcal{T}_{\Sigma}(Var)}{\mathcal{A} \vdash \theta(\varphi) \Rightarrow \theta(\varphi')}$$

Transitivity

$$\frac{\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi_2 \qquad \mathcal{A} \vdash \varphi_2 \Rightarrow \varphi_3}{\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi_3}$$

More Formally: Proof System III

- Rules of deductive nature

Case analysis

$$\frac{\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi \qquad \mathcal{A} \vdash \varphi_2 \Rightarrow \varphi}{\mathcal{A} \vdash \varphi_1 \vee \varphi_2 \Rightarrow \varphi}$$

Logic framing

$$\frac{\mathcal{A} \vdash \varphi \Rightarrow \varphi' \qquad \psi \text{ is a FOL}_{=} \text{ formula}}{\mathcal{A} \vdash \varphi \wedge \psi \Rightarrow \varphi' \wedge \psi}$$

More Formally: Proof System IV

Consequence

$$\frac{\models \varphi_1 \rightarrow \varphi_1' \qquad \mathcal{A} \vdash \varphi_1' \Rightarrow \varphi_2' \qquad \models \varphi_2' \rightarrow \varphi_2}{\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi_2}$$

Abstraction

$$\frac{\mathcal{A} \vdash \varphi \Rightarrow \varphi' \qquad X \cap \mathit{FreeVars}(\varphi') = \emptyset}{\mathcal{A} \vdash \exists X\, \varphi \Rightarrow \varphi'}$$

More Formally: Proof System V

- Main proof rule of matching logic rewriting

Circularity

$$\frac{\mathcal{A} \vdash \varphi \Rightarrow^{+} \varphi'' \qquad \mathcal{A} \cup \{\varphi \Rightarrow \varphi'\} \vdash \varphi'' \Rightarrow \varphi'}{\mathcal{A} \vdash \varphi \Rightarrow \varphi'}$$

Fact

- Matching logic generalizes both operational semantics and axiomatic semantics
  - Operational semantics by means of capturing term rewriting as discussed above
  - Axiomatic semantics by noticing that Hoare triples are particular pattern rewrites:

$$\mathrm{HL2ML}(\{\psi\}\,\mathrm{s}\,\{\psi'\}) \quad =$$

$$\langle \mathtt{s},\ \sigma_Z\rangle \wedge \sigma_Z(\psi) \Rightarrow \exists Z(\langle \mathtt{skip},\ \sigma_Z\rangle \wedge \sigma_Z(\psi'))$$

Theorems

- Any operational behavior can also be derived using matching logic reasoning
- For any Hoare triple $\{\psi\}\,\mathrm{s}\,\{\psi'\}$ derived with axiomatic semantics, the corresponding matching logic rule $\mathrm{HL2ML}(\{\psi\}\,\mathrm{s}\,\{\psi'\})$ can be derived with the matching logic proof system
  - Proof is constructive, not existential
- Partial correctness
  - Holds for ALL languages

MatchC

- A Matching Logic Verifier for (a fragment of) C
- Uses K semantics of C fragment *unchanged*
- Has verified a series of challenging programs
  - Undefiness, typical Hoare-like programs, heap programs (lists, trees, stacks, queues, graphs), sortings, AVL trees, Schorr-Waite graph marking

Conclusions

- K (semantics) and Matching Logic (verification)
- Formal semantics is useful and practical!
- One can use an executable semantics of a language *as is* also for program verification
  - As opposed to redefining it as a Hoare logic
- Giving a formal semantics is not necessarily painful, it can be fun if one uses the right tools